Diagnosing and Fixing Memory Leaks in Web Applications: Tips from the Front Line

Mark Thomas, Staff Engineer

CONFIDENTIAL

Introduction

- **Mark Thomas**
- **Involved in Apache Tomcat for 7 years**
  - Wrote the first memory leak detection and prevention implementation for Tomcat
  - Implemented a large proportion of Servlet 3.0, JSP 2.2 & EL 2.2 for Tomcat 7
  - Currently the Tomcat 7.0.x release manager
  - Created Tomcat’s security pages
  - Committer, PMC member
- **Apache Software Foundation**
  - Member
  - Part of the infrastructure team
- **Staff Engineer at VMware**
  - Tomcat / httpd consulting and training
  - Lead the SpringSource security team

Agenda

- **How it all started**
- **How memory leaks occur**
- **Debugging a leak – demonstration**
- **Root causes of leaks**
  - Those already fixed
  - Future plans
- **Questions**

How it all started

How it all started

- **Presenting on Servlet 3.0 / Tomcat 7 to an audience like this**
- **Made an off-the-cuff remark**
  - “Permgen errors on reload are not caused by Tomcat bugs, they are caused by application bugs”
- **That generated a lot of discussion**
- **Spent the rest of the conference debugging memory leaks with attendees**
  - Tomcat wasn’t causing the leaks
  - Neither were the applications, at least not directly
  - Root cause often in JRE code, triggered by 3rd party library
- **Wrote some fixes for the specific issues seen**

How it all started

- **Patterns soon started to emerge**
- **Realised that Tomcat could provide generic fixes**
- **Start of what became: org.apache.catalina.core.JreMemoryLeakPreventionListener**
- **Then ran some tests with some leaky applications**
  - Spring sample applications
  - Test cases provided by users
  - A couple of internal web applications
- **Added additional detection and prevention based on these**
- **The user community has provided additional ideas and feedback**

How memory leaks occur

How memory leaks occur: A little theory

- **A class is uniquely identified by**
  - Its name
  - The class loader that loaded it
- **Hence, you can have a class with the same name loaded multiple times in a single JVM, each in a different class loader**
- **Web containers use this for isolating web applications**
- **Each web application gets its own class loader**
- **Web application A can use Spring 2.5.6 whilst web application B can use Spring 3.0.2 without any conflicts**
- **Other containers, e.g. OSGI, use a similar approach**
- **Classes are loaded into the Permanent Generation**

How memory leaks occur: Reference chains

- An object retains a reference to the class it is an instance of
- A class retains a reference to the class loader that loaded it
- The class loader retains a reference to every class it loaded

- Retaining a reference to a single object from a web application pins every class loaded by the web application in the Permanent Generation
- These references often remain after a web application reload
- With each reload, more classes get pinned in the Permanent Generation and eventually it fills up

Debugging a leak - demonstration

Apache Tomcat 7, YourKit Java Profiler, Simple web application

Debugging memory leaks

- **Reload the application once**
- **Force GC**
- **Look for org.apache.catalina.loader.WebappClassLoader instances**
- **There should be exactly one per deployed application**
- **If you have more than that**
  - look for the instance where started = false
  - trace its GC roots
  - that will tell you what is holding the reference
  - finding what created the reference might be harder
- **A profiler makes this easy**
- **There are lots of good profilers available**
  - Full disclosure: I use YourKit because they give me a free copy to use with Tomcat

Root causes

JRE triggered leaks

JRE triggered leaks

- **All take a similar form**
- **Singleton / static initialiser**
  - Can be a Thread
  - Something that won't get garbage collected
- **Retains a reference to the context class loader when loaded**
- **If web application code triggers the initialisation**
  - The context class loader will be web application class loader
  - A reference is created to the web application class loader
  - This reference is never garbage collected
  - Pins the class loader (and hence all the classes it loaded) in memory
- **Prevented by the JreMemoryLeakPreventionListener**

JRE triggered leaks: sun.awt.AppContext

- **Triggered by**
  - Use of javax.imageio (e.g. Google Web Toolkit)
  - Use of java.beans.Introspector.flushCaches()
    - Ironically, Tomcat calls this to try and prevent memory leaks through the bean cache
  - Probably many others
- **Prevented in Tomcat by:**
  - Calling ImageIO.getCacheDirectory()
  - Pins Tomcat’s common class loader in memory
  - This is fine – don’t expect to throw this one away
  - Might be different if embedding Tomcat

JRE triggered leaks: sun.misc.GC.requestLatency(long)

- **Starts a GC Daemon thread**
- **Thread's context class loader will be context class loader when thread is started**
- **Triggered by:**
  - javax.management.remote.rmi.RMIConnectorServer.start()
  - Possibly others
- **Prevented in Tomcat by:**
  - Calling sun.misc.GC.requestLatency(long)
  - Has to use reflection
  - JVM specific so need to handle other JVMs
  - Pins Tomcat's class loader in memory
  - Should be OK (remember embedding)

JRE triggered leaks: More threads

- **Both very similar to previous slide**
- **sun.net.www.http.HttpClient**
  - Starts an HTTP keep-alive thread
  - Triggered by URL. openConnection()
  - Prevented in Tomcat by loading the sun.net.www.http.HttpClient class
  - JVM specific
- **Java Cryptography Architecture**
  - Starts a Token poller thread
  - Triggered by creating a message digest (under certain conditions)
  - Prevented in Tomcat by calling java.security.Security.getProviders();

JRE triggered leaks: JarURLConnections

- **URL connections are cached by default**
- **An open JarURLConnection locks the JAR file**
- **Affects all operating systems**
  - Harder to ignore on Windows
  - Prevents web applications from being undeployed
  - Potential security risk
- **Triggered by**
  - log4j 1.2.15 and earlier
  - javax.xml.bind.JAXBContext.newInstance()
- **Prevented in Tomcat by:**
  - Disable caching by default

JRE triggered leaks: XML parsing

- **Don't know why this triggers a leak**
- **No GC roots reported by profilers**
  - JVM bug
  - http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6916498
- **Made it very difficult to track down**
- **Triggered by:**
  - DocumentBuilderFactory.newInstance();
- **Prevented in Tomcat by:**
  - DocumentBuilderFactory.newInstance();

Root causes

Application triggered leaks

Application triggered leaks

- **All take a similar form**
- **Application registers an object with a JRE provided registry**
- **JRE registry is loaded by the system class loader**
- **Not cleared when web application is reloaded**
- **Reference chain**
  - Registry retains a reference to the object
  - Object retains a reference to its class
  - Class retains a reference to its class loader (web application class loader)
  - Class loader retains references to all classes it loaded

- **Applications are responsible for clearing references they create**
- **Failure to do so is logged on application stop**

Application triggered leaks: JDBC drivers

- **JDBC drivers are automatically registered with java.sql.DriverManager**
  - When loaded
  - Through the services API
- **JDBC drivers are NOT automatically de-registered**
- **Applications must de-register JDBC drivers when stopped**
- **Use a javax.servlet.ServletContextListener**
  - contextDestroyed() event
- **Tomcat will de-register JDBC drivers if the applications forgets**

Application triggered leaks: Threads

- **Threads started by a web application will have the web application class loader as the context class loader**
- **Applications must stop threads they start**
- **Tomcat will log an error if applications forget**
- **Tomcat can try and stop the thread (requires configuration)**
  - TimerThread via reflection – fairly safe
  - If started via an Executor via reflection– fairly safe
  - Thread.stop() – unsafe
- **Stopping threads**
  - Code is not thread safe
  - Often causes a JVM crash

Application triggered leaks: ThreadLocals

- **The lifecycle of a ThreadLocal must match that of a request**
- **An application may never see a Thread again**
  - No way to remove the ThreadLocal later
- **Applications must clear any ThreadLocals they create in the same request**
- **Tomcat will log an error if applications forget**
- **Tomcat can try and clear the ThreadLocal (requires configuration)**
  - Code is not thread safe
  - Not seen a problem in testing

Application triggered leaks: Non-application issues

- **sun.rmi.transport.Target**
  - Nothing the application can do to clear these
  - Tomcat does it silently via reflection
- **ResourceBundle**
  - Uses a weak reference
  - Still appears to trigger leaks
  - Tomcat clears the references silently via reflection
- **static final reference**
  - Not cleared by GC in some (very) old JVMs
  - Code still present
  - Disabled by default in Tomcat 7
- **Tomcat also clears references it creates**
  - loggers, introspection utils

Future plans

Future plans

- **See https://issues.apache.org/bugzilla**
- **Bugs**
  - Leaks triggered by JSP pages aren't detected or cleared (48837)
- **Enhancements**
  - Generic solution to ThreadLocal issues
    Renew the thread pool after application reload (49159)
  - Add the start date when reporting leaks in the manager app (49395)
- **Can we reduce the leak by somehow manipulating the class loader?**
  - No success so far
- **Keep 6.0.x in sync with new features as they are added to 7.0.x**
  - 6.0.30 will have all the latest changes

Useful links

Useful links

- **http://tomcat.apache.org/**
- **http://svn.apache.org/viewvc/tomcat/trunk/java/org/apache/**
  catalina/core/JreMemoryLeakPreventionListener.java
  catalina/loader/WebappClassLoader.java
- **http://wiki.apache.org/tomcat/MemoryLeakProtection**

- **Mailing lists**
  - announce@tomact.apache.org
  - users@tomcat.apache.org
  - dev@tomcat.apache.org

Questions